

**SR310871671**

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

**APPLICATION FOR LETTERS PATENT**

**System and Method for Transferring Computer-  
Readable Objects Across a Remote Boundary**

Inventor(s):

Jeffrey P. Snover

Daryl W. Wray

Rajesh Chandrashekar

Shankara Shastry

Hitesh Raigandhi

ATTORNEY'S DOCKET NO. MS1-1902US

1 **TECHNICAL FIELD**

2 Subject matter disclosed herein relates to remote computing systems, and  
3 in particular to methods for transferring computer-readable objects during remote  
4 communications.  
5

6 **BACKGROUND OF THE INVENTION**

7 Today, during software development, developers utilize a programming  
8 concept called object-oriented programming. In object-oriented programming,  
9 computer-readable objects (objects) are defined. The objects have properties and  
10 methods. The properties store data that pertains to the object. The methods  
11 perform functionality associated with the object and typically, provide interfaces  
12 to the properties defined in the object. Even though programming with objects  
13 provides great versatility, using objects across remote boundaries, such as between  
14 different computers, presents some challenges.

15 One challenge is determining the mechanism for transferring the objects  
16 between the two computers. In certain environments, the executable code  
17 (methods) for an object, along with its properties, are transferred to a requesting  
18 computer from a server computer. However, this solution may pose a security risk  
19 to the requesting computer if the executable code that is transferred performs a  
20 malicious action, such as deleting files. Thus, other solutions have been  
21 developed to minimize this potential risk.

22 One current solution is a technology called Web Services Technology.  
23 Using this technology, the object is converted into XML (extended mark-up  
24 language), which is transmitted to the requesting computer. Upon receiving the  
25

1 XML, the requesting computer converts the XML back into the object. The  
2 requesting computer may then access the object's properties and invoke the  
3 object's methods. Using this technology, the requesting computer is responsible  
4 for knowing and trusting the server from which the object is received. In addition,  
5 for any object that wishes to communicate with the requesting computer, the  
6 software developer, who developed the object, must implement a special interface  
7 to handle the communication. The special interface provides a mechanism for  
8 serializing and de-serializing the object in order for the object to be transferred.

9 Even though this Web Service solution provides a robust environment for  
10 transferring objects across remote boundaries, the technology is restrictive and  
11 burdensome. In some environments, such as in system administration  
12 environments, forcing software developers to implement a special interface for  
13 objects that system administration tasks wish to monitor, is not a viable solution.  
14 For example, requiring the developers to implement these special interfaces is not  
15 a trivial matter, and requires the developers to divert their attention away from  
16 their primary objective - implementing the object for their own particular  
17 application. Thus, many developers do not implement the special interfaces for  
18 their objects, and thus, these objects are not accessible.

19 Therefore, there is a need for a method of transferring objects across a  
20 remote boundary that is secure, not restrictive, and not burdensome to software  
21 developers.

## 22 **SUMMARY OF THE INVENTION**

23 The invention is directed to mechanisms and techniques for remote  
24 communication of objects. Briefly stated, a system and method for securely  
25

transferring computer-readable objects across a remote boundary is provided. The method decomposes any type of object into a hierarchy of sub-components based on a list of known object types. Each sub-component either corresponds to a known object type or an unknown object type. The unknown object types may be decomposed further into known object types at another level in the hierarchy. The known objects in the hierarchy are serialized into a package that is transmitted to a remote entity. The remote entity reconstructs the hierarchy. For any of the known object types, the remote entity instantiates an object of the known object type and populates the object with information transmitted in the package. The decomposition may be limited by specifying a level for the hierarchy, specifying a number that limits the known objects that are serialized, or specifying the properties within the object to serialize.

#### **BRIEF DESCRIPTION OF THE DRAWINGS**

FIGURE 1 illustrates an exemplary computing device on which an exemplary administrative tool framework may be implemented.

FIGURE 2 is a block diagram generally illustrating an overview of an exemplary administrative tool framework.

FIGURE 3 is a block diagram illustrating components within the host-specific components of the administrative tool framework shown in FIGURE 2.

FIGURE 4 is a block diagram illustrating components within the core engine component of the administrative tool framework shown in FIGURE 2.

FIGURE 5 is a functional block diagram of an exemplary extended type manager suitable for use within the administrative tool framework shown in FIGURE 2.

1       FIGURE 6 is an exemplary data structure for specifying a cmdlet suitable  
2 for use within the administrative tool framework shown in FIGURE 2.

3       FIGURE 7 is a functional block diagram illustrating components within the  
4 administrative tool framework shown in FIGURE 2 for performing remote  
5 processing of cmdlets.

6       FIGURE 8 is a logical flow diagram illustrating an exemplary process for  
7 processing a cmdlet.

8       FIGURE 9 is a block diagram illustrating an overview of the serialization  
9 process and de-serialization process suitable for use in FIGURE 8.

10       FIGURE 10 is a logical flow diagram illustrating an exemplary process for  
11 serializing objects suitable for use in the processing of the cmdlet shown in  
12 FIGURE 8.

13       FIGURE 11 is a logical flow diagram illustrating an exemplary process for  
14 de-serializing objects suitable for use in the processing of the cmdlet shown in  
15 FIGURE 8.

16       FIGURE 12 is a graphical representation of a property bag hierarchy  
17 generated during the serialization process shown in FIGURE 10.

18       FIGURE 13 is a logical flow diagram illustrating an exemplary process for  
19 negotiating a protocol suitable for use in conjunction with the process for  
20 serializing objects shown in FIGURE 10.

## 21       **DETAILED DESCRIPTION**

22       Briefly stated, the present system and method for transferring objects across  
23 a remote boundary provides a secure method for transferring objects. In addition,  
24 the present system and method does not place any artificial requirements on  
25 objects. Therefore, software developers do not incur any burden for supporting

1 remote operations using their objects. Thus, unlike existing systems, any object  
2 on any computer or in another process on the same computer may be transferred  
3 across the remote boundary to a requesting process. In addition, the present  
4 system and method for transferring objects does not require both computers to  
5 execute the same version of software. Rather, the method incorporates a protocol  
6 negotiation process that not only provides a mechanism for supporting  
7 communication between two different versions, but also minimizes the amount of  
8 data transferred over the remote boundary.

9 The following detailed description is divided into several sections. In  
10 general, the present system and method is described within the context of an  
11 exemplary administrative tool environment. However, after reading the following  
12 description, those skilled in the art will appreciate that the present method may be  
13 implemented in other exemplary environments, which are also included within the  
14 scope of the appended claims.

15 A first section describes an illustrative computing environment in which an  
16 exemplary administrative tool environment may operate. A second section  
17 describes an exemplary framework for the administrative tool environment.  
18 Subsequent sections describe individual components of the exemplary framework  
19 and the operation of these components. For example, the section on "Exemplary  
20 Remote Processing of a Cmdlet", in conjunction with FIGURES 7-13, describes  
21 an exemplary system and method for transferring objects across a remote  
22 boundary.

### 23 Exemplary Computing Environment

24 FIGURE 1 illustrates an exemplary computing device that may be used in  
25 an exemplary administrative tool environment. In a very basic configuration,

1 computing device 100 typically includes at least one processing unit 102 and  
2 system memory 104. Depending on the exact configuration and type of  
3 computing device, system memory 104 may be volatile (such as RAM), non-  
4 volatile (such as ROM, flash memory, etc.) or some combination of the two.  
5 System memory 104 typically includes an operating system 105, one or more  
6 program modules 106, and may include program data 107. The operating system  
7 106 include a component-based framework 120 that supports components  
8 (including properties and events), objects, inheritance, polymorphism, reflection,  
9 and provides an object-oriented component-based application programming  
10 interface (API), such as that of the .NET<sup>TM</sup> Framework manufactured by Microsoft  
11 Corporation, Redmond, WA. The operating system 105 also includes an  
12 administrative tool framework 200 that interacts with the component-based  
13 framework 120 to support development of administrative tools (not shown). This  
14 basic configuration is illustrated in FIGURE 1 by those components within dashed  
15 line 108.

16 Computing device 100 may have additional features or functionality. For  
17 example, computing device 100 may also include additional data storage devices  
18 (removable and/or non-removable) such as, for example, magnetic disks, optical  
19 disks, or tape. Such additional storage is illustrated in FIGURE 1 by removable  
20 storage 109 and non-removable storage 110. Computer storage media may  
21 include volatile and nonvolatile, removable and non-removable media  
22 implemented in any method or technology for storage of information, such as  
23 computer readable instructions, data structures, program modules, or other data.  
24 System memory 104, removable storage 109 and non-removable storage 110 are  
25

1 all examples of computer storage media. Computer storage media includes, but is  
2 not limited to, RAM, ROM, EEPROM, flash memory or other memory  
3 technology, CD-ROM, digital versatile disks (DVD) or other optical storage,  
4 magnetic cassettes, magnetic tape, magnetic disk storage or other magnetic storage  
5 devices, or any other medium which can be used to store the desired information  
6 and which can be accessed by computing device 100. Any such computer storage  
7 media may be part of device 100. Computing device 100 may also have input  
8 device(s) 112 such as keyboard, mouse, pen, voice input device, touch input  
9 device, etc. Output device(s) 114 such as a display, speakers, printer, etc. may  
10 also be included. These devices are well known in the art and need not be  
11 discussed at length here.

12 Computing device 100 may also contain communication connections 116  
13 that allow the device to communicate with other computing devices 118, such as  
14 over a network. Communication connections 116 are one example of  
15 communication media. Communication media may typically be embodied by  
16 computer readable instructions, data structures, program modules, or other data in  
17 a modulated data signal, such as a carrier wave or other transport mechanism, and  
18 includes any information delivery media. The term "modulated data signal"  
19 means a signal that has one or more of its characteristics set or changed in such a  
20 manner as to encode information in the signal. By way of example, and not  
21 limitation, communication media includes wired media such as a wired network or  
22 direct-wired connection, and wireless media such as acoustic, RF, infrared and  
23 other wireless media. The term computer readable media as used herein includes  
24 both storage media and communication media.  
25



## Exemplary Administrative Tool Framework

FIGURE 2 is a block diagram generally illustrating an overview of an exemplary administrative tool framework 200. Administrative tool framework 200 includes one or more host components 202, host-specific components 204, host-independent components 206, and handler components 208. The host-independent components 206 may communicate with each of the other components (i.e., the host components 202, the host-specific components 204, and the handler components 208). Each of these components are briefly described below and described in further detail, as needed, in subsequent sections.

### Host components

The host components 202 include one or more host programs (e.g., host programs 210-214) that expose automation features for an associated application to users or to other programs. Each host program 210-214 may expose these automation features in its own particular style, such as via a command line, a graphical user interface (GUI), a voice recognition interface, application programming interface (API), a scripting language, a web service, and the like. However, each of the host programs 210-214 expose the one or more automation features through a mechanism provided by the administrative tool framework.

In this example, the mechanism uses cmdlets to surface the administrative tool capabilities to a user of the associated host program 210-214. In addition, the mechanism uses a set of interfaces made available by the host to embed the administrative tool environment within the application associated with the corresponding host program 210-214. Throughout the following discussion, the

1 term "cmdlet" is used to refer to commands that are used within the exemplary  
2 administrative tool environment.

3 Cmdlets correspond to commands in traditional administrative  
4 environments. However, cmdlets are quite different than these traditional  
5 commands. For example, cmdlets are typically smaller in size than their  
6 counterpart commands because the cmdlets can utilize common functions  
7 provided by the administrative tool framework, such as parsing, data validation,  
8 error reporting, and the like. Because such common functions can be implemented  
9 once and tested once, the use of cmdlets throughout the administrative tool  
10 framework allows the incremental development and test costs associated with  
11 application-specific functions to be quite low compared to traditional  
12 environments.

13 In addition, in contrast to traditional environments, cmdlets do not need to  
14 be stand-alone executable programs. Rather, cmdlets may run in the same  
15 processes within the administrative tool framework. This allows cmdlets to  
16 exchange "live" objects between each other. This ability to exchange "live"  
17 objects allows the cmdlets to directly invoke methods on these objects.

18 In overview, each host program 210-214 manages the interactions between  
19 the user and the other components within the administrative tool framework.  
20 These interactions may include prompts for parameters, reports of errors, and the  
21 like. Typically, each host program 210-213 may provide its own set of specific  
22 host cmdlets (e.g., host cmdlets 218). For example, if the host program is an email  
23 program, the host program may provide host cmdlets that interact with mailboxes  
24 and messages. Even though FIGURE 2 illustrates host programs 210-214, one  
25

1 skilled in the art will appreciate that host components 202 may include other host  
2 programs associated with existing or newly created applications. These other host  
3 programs will also embed the functionality provided by the administrative tool  
4 environment within their associated application.

5 In the examples illustrated in FIGURE 2, a host program may be a  
6 management console (i.e., host program 210) that provides a simple, consistent,  
7 administration user interface for users to create, save, and open administrative  
8 tools that manage the hardware, software, and network components of the  
9 computing device. To accomplish these functions, host program 210 provides a  
10 set of services for building management GUIs on top of the administrative tool  
11 framework. The GUI interactions may also be exposed as user-visible scripts that  
12 help teach the users the scripting capabilities provided by the administrative tool  
13 environment.

14 In another example, the host program may be a command line interactive  
15 shell (i.e., host program 212). The command line interactive shell may allow shell  
16 metadata 216 to be input on the command line to affect processing of the  
17 command line.

18 In still another example, the host program may be a web service (i.e., host  
19 program 214) that uses industry standard specifications for distributed computing  
20 and interoperability across platforms, programming languages, and applications.  
21 In another example, illustrated in FIGURE 7, the host program may provide a  
22 remote interface for communicating with a remote computer.

23  
24 In addition to these examples, third parties may add their own host  
25 components by creating “third party” or “provider” interfaces and provider

1 cmdlets that are used with their host program or other host programs. The  
2 provider interface exposes an application or infrastructure so that the application  
3 or infrastructure can be manipulated by the administrative tool framework. The  
4 provider cmdlets provide automation for navigation, diagnostics, configuration,  
5 lifecycle, operations, and the like. The provider cmdlets exhibit polymorphic  
6 cmdlet behavior on a completely heterogeneous set of data stores. The  
7 administrative tool environment operates on the provider cmdlets with the same  
8 priority as other cmdlet classes. The provider cmdlet is created using the same  
9 mechanisms as the other cmdlets. The provider cmdlets expose specific  
10 functionality of an application or an infrastructure to the administrative tool  
11 framework. Thus, through the use of cmdlets, product developers need only create  
12 one host component that will then allow their product to operate with many  
13 administrative tools. For example, with the exemplary administrative tool  
14 environment, system level graphical user interface help menus may be integrated  
15 and ported to existing applications.

### 16 **Host-specific components**

17 The host-specific components 204 include a collection of services that  
18 computing systems (e.g., computing device 100 in FIGURE 1) use to isolate the  
19 administrative tool framework from the specifics of the platform on which the  
20 framework is running. Thus, there is a set of host-specific components for each  
21 type of platform. The host-specific components allow the users to use the same  
22 administrative tools on different operating systems.

23 Turning briefly to FIGURE 3, the host-specific components 204 may  
24 include an intellisense/metadata access component 302, a help cmdlet component  
25

1 304, a configuration/registration component 306, a cmdlet setup component 308,  
2 and an output interface component 309. Components 302-308 communicate with a  
3 database store manager 312 associated with a database store 314. The parser 220  
4 and script engine 222 communicate with the intellisense/metadata access  
5 component 302. The core engine 224 communicates with the help cmdlet  
6 component 304, the configuration/registration component 306, the cmdlet setup  
7 component 308, and the output interface component 309. The output interface  
8 component 309 includes interfaces provided by the host to out cmdlets. These out  
9 cmdlets can then call the host's output object to perform the rendering. Host-  
10 specific components 204 may also include a logging/auditing component 310,  
11 which the core engine 224 uses to communicate with host specific (i.e., platform  
12 specific) services that provide logging and auditing capabilities.

13 In one exemplary administrative tool framework, the intellisense/metadata  
14 access component 302 provides auto-completion of commands, parameters, and  
15 parameter values. The help cmdlet component 304 provides a customized help  
16 system based on a host user interface.

### 17 **Handler components**

18 Referring back to FIGURE 2, the handler components 208 includes legacy  
19 utilities 230, management cmdlets 232, non-management cmdlets 234, remoting  
20 cmdlets 236, and a web service interface 238. The management cmdlets 232 (also  
21 referred to as platform cmdlets) include cmdlets that query or manipulate the  
22 configuration information associated with the computing device. Because  
23 management cmdlets 232 manipulate system type information, they are dependant  
24 upon a particular platform. However, each platform typically has management  
25

cmdlets 232 that provide similar actions as management cmdlets 232 on other platforms. For example, each platform supports management cmdlets 232 that get and set system administrative attributes (e.g., get/process, set/IPAddress). The host-independent components 206 communicate with the management cmdlets via cmdlet objects generated within the host-independent components 206.

The non-management cmdlets 234 (sometimes referred to as base cmdlets) include cmdlets that group, sort, filter, and perform other processing on objects provided by the management cmdlets 232. The non-management cmdlets 234 may also include cmdlets for formatting and outputting data associated with the pipelined objects. The non-management cmdlets 234 may be the same on each platform and provide a set of utilities that interact with host-independent components 206 via cmdlet objects. The interactions between the non-management cmdlets 234 and the host-independent components 206 allow reflection on objects and allow processing on the reflected objects independent of their (object) type. Thus, these utilities allow developers to write non-management cmdlets once and then apply these non-management cmdlets across all classes of objects supported on a computing system. In the past, developers had to first comprehend the format of the data that was to be processed and then write the application to process only that data. As a consequence, traditional applications could only process data of a very limited scope.

The legacy utilities 230 include existing executables, such as win32 executables that run under cmd.exe. Each legacy utility 230 communicates with the administrative tool framework using text streams (i.e., stdin and stdout), which are a type of object within the object framework. Because the legacy utilities 230

1 utilize text streams, reflection-based operations provided by the administrative tool  
2 framework are not available. The legacy utilities 230 execute in a different  
3 process than the administrative tool framework. Although not shown, other  
4 cmdlets may also operate out of process.

5 The remoting cmdlets 236, in combination with the web service interface  
6 238, provide remoting mechanisms to access interactive and programmatic  
7 administrative tool environments on other computing devices over a  
8 communication media, such as internet or intranet (e.g., internet/intranet 240  
9 shown in FIGURE 2). In one exemplary administrative tool framework, the  
10 remoting mechanisms support federated services that depend on infrastructure that  
11 spans multiple independent control domains. The remoting mechanism allows  
12 scripts to execute on remote computing devices. The scripts may be run on a  
13 single or on multiple remote systems. The results of the scripts may be processed  
14 as each individual script completes or the results may be aggregated and processed  
15 en-masse after all the scripts on the various computing devices have completed.

16 For example, web service 214 shown as one of the host components 202  
17 may be a remote agent. The remote agent handles the submission of remote  
18 command requests to the parser and administrative tool framework on the target  
19 system. The remoting cmdlets serve as the remote client to provide access to the  
20 remote agent. The remote agent and the remoting cmdlets communicate via a  
21 parsed stream. This parsed stream may be protected at the protocol layer, or  
22 additional cmdlets may be used to encrypt and then decrypt the parsed stream.  
23 Exemplary components for processing remote cmdlets and transferring objects  
24  
25

1 between remote processes is illustrated in FIGURE 7 and described below in the  
2 section entitled "Exemplary Remote Processing of a Cmdlet".

### 3 **Host-independent components**

4 The host-independent components 206 include a parser 220, a script engine  
5 222 and a core engine 224. The host-independent components 206 provide  
6 mechanisms and services to group multiple cmdlets, coordinate the operation of  
7 the cmdlets, and coordinate the interaction of other resources, sessions, and jobs  
8 with the cmdlets.

### 9 **Exemplary Parser**

10 The parser 220 provides mechanisms for receiving input requests from  
11 various host programs and mapping the input requests to uniform cmdlet objects  
12 that are used throughout the administrative tool framework, such as within the  
13 core engine 224. In addition, the parser 220 may perform data processing based  
14 on the input received. The parser 220 of the present administrative tool  
15 framework provides the capability to easily expose different languages or syntax  
16 to users for the same capabilities. For example, because the parser 220 is  
17 responsible for interpreting the input requests, a change to the code within the  
18 parser 220 that affects the expected input syntax will essentially affect each user of  
19 the administrative tool framework. Therefore, system administrators may provide  
20 different parsers on different computing devices that support different syntax.  
21 However, each user operating with the same parser will experience a consistent  
22 syntax for each cmdlet. In contrast, in traditional environments, each command  
23 implemented its own syntax. Thus, with thousands of commands, each  
24  
25



environment supported several different syntax, usually many of which were inconsistent with each other.

### **Exemplary Script Engine**

The script engine 222 provides mechanisms and services to tie multiple cmdlets together using a script. A script is an aggregation of command lines that share session state under strict rules of inheritance. The multiple command lines within the script may be executed either synchronously or asynchronously, based on the syntax provided in the input request. The script engine 222 has the ability to process control structures, such as loops and conditional clauses and to process variables within the script. The script engine also manages session state and gives cmdlets access to session data based on a policy (not shown).

### **Exemplary Core Engine**

The core engine 224 is responsible for processing cmdlets identified by the parser 220. Turning briefly to FIGURE 4, an exemplary core engine 224 within the administrative tool framework 200 is illustrated. The exemplary core engine 224 includes a pipeline processor 402, a loader 404, a metadata processor 406, an error & event handler 408, a session manager 410, and an extended type manager 412.

### **Exemplary Metadata Processor**

The metadata processor 406 is configured to access and store metadata within a metadata store, such as database store 314 shown in FIGURE 3. The metadata may be supplied via the command line, within a cmdlet class definition, and the like. Different components within the administrative tool framework 200

1 may request the metadata when performing their processing. For example, parser  
2 202 may request metadata to validate parameters supplied on the command line.

### 3 Exemplary Error & Event Processor

4 The error & event processor 408 provides an error object to store  
5 information about each occurrence of an error during processing of a command  
6 line. For additional information about one particular error and event processor  
7 which is particularly suited for the present administrative tool framework, refer to  
8 U.S. Patent Application No. 10/413,054, entitled "System and Method for  
9 Persisting Error Information in a Command Line Environment", which is owned  
10 by the same assignee as the present invention, and is incorporated here by  
11 reference.

### 12 Exemplary Session Manager

13 The session manager 410 supplies session and state information to other  
14 components within the administrative tool framework 200. The state information  
15 managed by the session manager may be accessed by any cmdlet, host, or core  
16 engine via programming interfaces. These programming interfaces allow for the  
17 creation, modification, and deletion of state information.

### 18 Exemplary Pipeline Processor and Loader

19 The loader 404 is configured to load each cmdlet in memory in order for  
20 the pipeline processor 402 to execute the cmdlet. The pipeline processor 402  
21 includes a cmdlet processor 420 and a cmdlet manager 422. The cmdlet  
22 processor 420 dispatches individual cmdlets. If the cmdlet requires execution on a  
23 remote, or a set of remote machines, the cmdlet processor 420 coordinates the  
24 execution with the remoting cmdlet 236 shown in FIGURE 2. The cmdlet  
25 manager 422 handles the execution of aggregations of cmdlets. The cmdlet

1 manager 422, the cmdlet processor 420, and the script engine 222 (FIGURE 2)  
2 communicate with each other in order to perform the processing on the input  
3 received from the host program 210-214. The communication may be recursive in  
4 nature. For example, if the host program provides a script, the script may invoke  
5 the cmdlet manager 422 to execute a cmdlet, which itself may be a script. The  
6 script may then be executed by the script engine 222.

### 7 Exemplary Extended Type Manager

8 As mentioned above, the administrative tool framework provides a set of  
9 utilities that allows reflection on objects and allows processing on the reflected  
10 objects independent of their (object) type. The administrative tool framework 200  
11 interacts with the component framework on the computing system (component  
12 framework 120 in FIGURE 1) to perform this reflection. As one skilled in the art  
13 will appreciate, reflection provides the ability to query an object and to obtain a  
14 type for the object, and then reflect on various objects and properties associated  
15 with that type of object to obtain other objects and/or a desired value.

16 Even though reflection provides the administrative tool framework 200 a  
17 considerable amount of information on objects, traditionally reflection focuses on  
18 the type of object. For example, when a database datatable is reflected upon, the  
19 information that is returned is that the datatable has two properties: a column  
20 property and a row property. These two properties do not provide sufficient detail  
21 regarding the “objects” within the datatable. Similar problems arise when  
22 reflection is used on extensible markup language (XML) and other objects.

23 In contrast, extended type manager 412 focuses on the usage of the type  
24 rather than the type of object. Thus, with this as its focus, the extended type  
25 manager determines whether the object can be used to obtain the required

1 information. Continuing with the above datatable example, knowing that the  
2 datatable has a column property and a row property is not particularly interesting.  
3 However, by focusing on the usage, the extended type manager associates each  
4 row with an “object” and associates each column with a “property” of that  
5 “object”. Thus, the extended type manager 412 provides a mechanism to create  
6 “objects” from any type of precisely parse-able input. In so doing, the extended  
7 type manager 412 supplements the reflection capabilities provided by the  
8 component-based framework 120 and extends “reflection” to any type of precisely  
9 parse-able input.

10 In overview, the extended type manager is configured to access precisely  
11 parse-able input (not shown) and to correlate the precisely parse-able input with a  
12 requested data type. The extended type manager 412 then provides the requested  
13 information to the requesting component, such as the pipeline processor 402 or  
14 parser 220. In the following discussion, precisely parse-able input is defined as  
15 input in which properties and values may be discerned. Some exemplary precisely  
16 parse-able input include Windows Management Instrumentation (WMI) input,  
17 ActiveX Data Objects (ADO) input, eXtensible Markup Language (XML) input,  
18 and object input, such as .NET objects. Other precisely parse-able input may  
19 include third party data formats.

20 FIGURE 5 is a functional block diagram of an exemplary extended type  
21 manager for use within the administrative tool framework. For explanation  
22 purposes, the functionality (denoted by the number “3” within a circle) provided  
23 by the extended type manager is contrasted with the functionality provided by a  
24 traditional tightly bound system (denoted by the number “1” within a circle) and  
25

1 the functionality provided by a reflection system (denoted by the number "2"  
2 within a circle). In the traditional tightly bound system, a caller 502 within an  
3 application directly accesses the information (e.g., properties P1 and P2, methods  
4 M1 and M2) within object A. As mentioned above, the caller 502 must know, a  
5 priori, the properties (e.g., properties P1 and P2) and methods (e.g., methods M1  
6 and M2) provided by object A at compile time. In the reflection system, generic  
7 code 520 (not dependent on any data type) queries a system 508 that performs  
8 reflection 510 on the requested object and returns the information (e.g., properties  
9 P1 and P2, methods M1 and M2) about the object (e.g., object A) to the generic  
10 code 520. Although not shown in object A, the returned information may include  
11 additional information, such as vendor, file, date, and the like. Thus, through  
12 reflection, the generic code 520 obtains at least the same information that the  
13 tightly bound system provides. The reflection system also allows the caller 502 to  
14 query the system and get additional information without any a priori knowledge of  
15 the parameters.

16 In both the tightly bound systems and the reflection systems, new data  
17 types can not be easily incorporated within the operating environment. For  
18 example, in a tightly bound system, once the operating environment is delivered,  
19 the operating environment can not incorporate new data types because it would  
20 have to be rebuilt in order to support them. Likewise, in reflection systems, the  
21 metadata for each object class is fixed. Thus, incorporating new data types is not  
22 usually done.

23 However, with the present extended type manager new data types can be  
24 incorporated into the operating system. With the extended type manager 522,  
25 generic code 520 may reflect on a requested object to obtain extended data types

1 (e.g., object A') provided by various external sources, such as a third party objects  
2 (e.g., object A' and B), a semantic web 532, an ontology service 534, and the like.  
3 As shown, the third party object may extend an existing object (e.g., object A') or  
4 may create an entirely new object (e.g., object B).

5 Each of these external sources may register their unique structure within a  
6 type metadata 540 and may provide code 542. When an object is queried, the  
7 extended type manager reviews the type metadata 540 to determine whether the  
8 object has been registered. If the object is not registered within the type metadata  
9 540, reflection is performed. Otherwise, extended reflection is performed. The  
10 code 542 returns the additional properties and methods associated with the type  
11 being reflected upon. For example, if the input type is XML, the code 542 may  
12 include a description file that describes the manner in which the XML is used to  
13 create the objects from the XML document. Thus, the type metadata 540  
14 describes how the extended type manager 412 should query various types of  
15 precisely parse-able input (e.g., third party objects A' and B, semantic web 532) to  
16 obtain the desired properties for creating an object for that specific input type and  
17 the code 542 provides the instructions to obtain these desired properties. As a  
18 result, the extended type manager 412 provides a layer of indirection that allows  
19 "reflection" on all types of objects. An exemplary implementation for this layer of  
20 indirection is described in greater detail below in conjunction with remote  
21 processing. In this implementation, a precisely parse-able input (e.g., a serialized  
22 object) is used to obtain the desired properties for creating an object for a specific  
23 input type (e.g., a property bag). The property bag is then further de-composed to  
24 create objects of one or more base types. As will be described, the extended type  
25 manager enables one embodiment for transferring objects between remote

1 processes and allows any object of any type to be transferred between the  
2 processes.

3 In addition to providing extended types, the extend type manager 412  
4 provides additional query mechanisms, such as a property path mechanism, a key  
5 mechanism, a compare mechanism, a conversion mechanism, a globber  
6 mechanism, a property set mechanism, a relationship mechanism, and the like.  
7 Various techniques may be used to implement the semantics for the extended type  
8 manager. Three techniques are described below. However, those skilled in the art  
9 will appreciate that variations of these techniques may be used.

10 In one technique, a series of classes having static methods (e.g.,  
11 `getProperty()` ) may be provided. An object is input into the static method (e.g.,  
12 `getProperty(object)` ), and the static method returns a set of results. In another  
13 technique, the operating environment envelopes the object with an adapter. Thus,  
14 no input is supplied. Each instance of the adapter has a `getProperty` method that  
15 acts upon the enveloped object and returns the properties for the enveloped object.  
16 The following is pseudo code illustrating this technique:

17  
18 Class Adaptor

19 {  
20       Object X;  
21       getProperties();  
22     }.  
23

24 In still another technique, an adaptor class subclasses the object.  
25 Traditionally, subclassing occurred before compilation. However, with certain

operating environments, subclassing may occur dynamically. For these types of environments, the following is pseudo code illustrating this technique:

```
Class Adaptor : A
{
    getProperties()
    {
        return data;
    }
}.
```

Thus, as illustrated in FIGURE 5, the extended type manager allows developers to create a new data type, register the data type, and allow other applications and cmdlets to use the new data type. In contrast, in prior administrative environments, each data type had to be known at compile time so that a property or method associated with an object instantiated from that data type could be directly accessed. Therefore, adding new data types that were supported by the administrative environment was seldom done in the past.

Referring back to FIGURE 2, in overview, the administrative tool framework 200 does not rely on the shell for coordinating the execution of commands input by users, but rather, splits the functionality into processing portions (e.g., host-independent components 206) and user interaction portions (e.g., via host cmdlets). In addition, the present administrative tool environment greatly simplifies the programming of administrative tools because the code required for parsing and data validation is no longer included within each



1 command, but is rather provided by components (e.g., parser 220) within the  
2 administrative tool framework. The exemplary processing performed within the  
3 administrative tool framework is described below.

### 4 **Exemplary Operation**

5 FIGURE 6 graphically illustrates an exemplary data structure used within  
6 the administrative tool environment. FIGURES 8-11 and 13 graphically illustrate  
7 exemplary processing flows within the administrative tool environment. One  
8 skilled in the art will appreciate that certain processing may be performed by a  
9 different component than the component described below without departing from  
10 the scope of the claims. Before describing the processing performed within the  
11 components of the administrative tool framework, an exemplary data structure for  
12 cmdlets used within the administrative tool framework is described.

### 13 **Exemplary Data Structures for Cmdlet Objects**

14 FIGURE 6 is an exemplary data structure for specifying a cmdlet suitable  
15 for use within the administrative tool framework shown in FIGURE 2. When  
16 completed, the cmdlet may be a management cmdlet, a non-management cmdlet, a  
17 host cmdlet, a provider cmdlet, or the like. The following discussion describes the  
18 creation of a cmdlet with respect to a system administrator's perspective (i.e., a  
19 provider cmdlet). However, each type of cmdlet is created in the same manner  
20 and operates in a similar manner. A cmdlet may be written in any language, such  
21 as C#. In addition, the cmdlet may be written using a scripting language or the  
22 like. When the administrative tool environment operates with the .NET  
23 Framework, the cmdlet may be a .NET object.

1       The provider cmdlet 600 (hereinafter, referred to as cmdlet 600) is a public  
2 class deriving from a cmdlet class 602. Cmdlet 600 includes a cmdlet class name  
3 specified in place of "<command name>". A software developer specifies a  
4 cmdletDeclaration 604 that associates a verb/noun pair 606, such as "get/process",  
5 "get/db", "format/table", and the like, with the cmdlet 600. The verb/noun pair  
6 606 is registered within the administrative tool environment. The verb or the noun  
7 may be implicit. The parser looks in the cmdlet registry to identify the cmdlet 600  
8 when a command string having the name (e.g., get/db) is supplied as input on a  
9 command line or in a script.

10       The cmdlet 600 is associated with a grammar mechanism that defines a  
11 grammar for expected input parameters to the cmdlet. The grammar mechanism  
12 may be directly or indirectly associated with the cmdlet. For example, cmdlet 600  
13 illustrates a direct grammar association. In cmdlet 600, one or more public  
14 parameters (e.g., Name 630 and State 632) are declared. Each public parameter  
15 630 and 632 may be associated with one or more types of attributes, such as input  
16 attribute 631 and 633. The input attributes 631 and 633 specify the manner in  
17 which public parameters 630 and 632, respectively, are populated. For example,  
18 the public parameter may be populated from an object emitted by another cmdlet  
19 in a pipeline of commands, from the command line, and the like. If the public  
20 parameters are populated from other objects, cmdlet 600 includes a first method  
21 640 (e.g., StartProcessing) and a second method 642 (e.g., processRecord). The  
22 core engine uses the first and second methods 640 and 642 to direct the processing  
23 of the cmdlet 600. For example, the first method 640 may be executed once and  
24 may perform set-up functions. The code within the second method 642 may be  
25

1 executed for each object (e.g., record) that needs to be processed by the cmdlet  
2 600. The cmdlet 600 may also include a third method (not shown) that cleans up  
3 after the cmdlet 600. Alternatively, the grammar mechanism may be indirectly  
4 associated with the cmdlet by using a description of the public parameters defined  
5 in an external source, such as an XML document. The description of the  
6 parameters in this external source would then drive the parsing of the input objects  
7 to the cmdlet.

8 Thus, as shown in FIGURE 6, code within the second method 642 is  
9 typically quite brief and does not contain functionality required in traditional  
10 administrative tool environments, such as parsing code, data validation code, and  
11 the like. Thus, system administrators can develop complex administrative tasks  
12 without learning a complex programming language.

13 The data structure 600 may also include a private member 650 that is not  
14 recognized as an input parameter. The private member 650 may be used for  
15 storing data that is generated within the cmdlet.

16 Exemplary process flows within the administrative tool environment are  
17 now described.  
18

### 19 **Exemplary Remote Processing of a Cmdlet**

20 FIGURE 7 is a functional block diagram generally illustrating a remote  
21 computing environment 700 that utilizes the mechanisms described in conjunction  
22 with the present invention. Illustrated is an "administrator" 710 computing system  
23 and a remote computing system (hereinafter referred to as remote server 720).  
24 The remote server 720 may be physically located anywhere and may be an  
25 individual computing system in use by an end user, such as an employee or

1 subscriber. While FIGURE 7 illustrates a single remote server 720, the remote  
2 computing environment 700 may include any number of remote servers. Each  
3 remote server performs processing as described below with reference to remote  
4 server 720.

5 The administrator 112 and remote server 720 may be computing devices,  
6 such as computing device 100 illustrated in FIGURE 1. The administrator 112 is  
7 used by a system administrator or the like to maintain the remote server 720. In  
8 other words, the administrator 112 runs commands and performs tasks that may  
9 query the status or state of the remote server 720 and/or make changes to the  
10 remote server 720. The administrator 112 includes an execution environment,  
11 which may include several components of the administrative tool framework 200  
12 shown in FIGURE 2. In particular, administrator 112 includes a remote interface  
13 714 and a list of base types 712. In one embodiment, the list of base types 712  
14 comprises a table having two columns: a first column for listing a type of object  
15 and a second column for referencing a specific serializer associated with the type.  
16 While other components within the administrative tool framework 200 may also  
17 be utilized to perform remote processing of cmdlets, the following discussion  
18 focuses on the interaction between the administrator 112 and the remote server 720  
19 in order to process a remote cmdlet, such as "Rcmd machine:RemoteServer  
20 Get/DB", and describes the operation of the other components, as needed, within  
21 this context. In addition, the following example illustrates remote processing of  
22 one cmdlet. However, the remote aspects work equally as well if there is a  
23 pipeline of cmdlets.

24 Remote server 720 includes a remote agent 724. The remote agent 724 is a  
25 component that responds to remote requests to execute one or more cmdlets. In

1 addition, remote agent 724 is configured to take the results of the execution of the  
2 one or more cmdlets and create a return package 730 that is returned to the  
3 requesting entity (e.g., administrator 112). In one implementation, the package  
4 takes the form of a serialized object that includes the results of execution, as well  
5 as meta information such as the date and time of invocation, identifying  
6 information about the particular remote system from which the results originated,  
7 and information about the requesting entity. This and perhaps other information is  
8 bound up into the return package 730 and transmitted back to the requesting entity  
9 (e.g., administrator 112).

10 It is important to note, that the requesting entity may be one process  
11 executing on a computing system and the remote server may be another process  
12 operating on the same computing system. In this configuration, the  
13 communication interface 740 includes system level application programming  
14 interfaces (API) for communicating between two processes, which are well known  
15 to those skilled in the art. In other embodiments, the communication interface 740  
16 includes an internet or intranet network.

17 FIGURE 8 is a logical flow diagram generally illustrating steps that may be  
18 performed by a process 800 for executing a cmdlet. The process begins at  
19 decision block 806, where a cmdlet has been input for execution, such as through  
20 the command line or the like. For example, the cmdlet, "Rcmd  
21 machine:RemoteServer Get/DB", may have been entered. At decision block 806,  
22 a location where the cmdlet is to be executed is determined in order to determine  
23 whether the execution crosses a remote boundary. In overview, each computing  
24 system (e.g., administrator 720, remote server 720) supports one or more  
25 processes. Each process hosts at least one program or application. In addition,

1 one process may host one or more application domains. Application domains are a  
2 relatively new mechanism that allows multiple applications to execute within the  
3 same process, yet still be isolated from other applications. The application domain  
4 is a logical and physical boundary created around an application by a runtime  
5 environment. Each application domain prevents the configuration, security, or  
6 stability of its respective application from affecting other applications in other  
7 application domains. Thus, the cmdlet may be executed in the following  
8 locations: 1) within the application domain of the requesting entity (e.g.,  
9 administrator); 2) within another application domain of the same process as the  
10 requesting entity; 3) within another process on the requesting entity; or 4) on a  
11 remote server. If the cmdlet executes either within another process (#3) or on a  
12 remote server (#4), the following discussion refers to this as executing over a  
13 “remote boundary”, which generally means executing outside a process boundary.

14 For specifying the location, in one embodiment, a parameter specified  
15 along with the cmdlet identifies which location to execute the cmdlet. For  
16 example, the parameter “-node” along with a relevant node name may indicate  
17 execution of the cmdlet on a remote server associated with the node name. The  
18 parameter “-workerprocess” may indicate execution of the cmdlet in another  
19 process of the requesting entity. The parameter “-appdomain” may indicate  
20 execution of the cmdlet within another application domain of the same process. If  
21 execution is to occur within the same application domain or within another  
22 application domain within the same process, processing continues at block 808.  
23 Otherwise processing continues at block 818. While the processing of blocks 808-  
24 814 is not essential to the understanding of remote processing, understanding this  
25

1 processing will aid in understanding the concerns that were overcome in order to  
2 support remote processing of cmdlets.

3 At block 808, a file associated with the cmdlet is identified. The  
4 identification of the cmdlet and its associated file may be thru registration. As  
5 described above in conjunction with FIGURE 6, the file contains the code for  
6 executing the cmdlet and also contains the properties and methods associated with  
7 the cmdlet class.

8 At block 810, the file is loaded into the process for the current application  
9 domain. As mentioned above, this blind loading of an executable file into the  
10 process on the requesting entity poses a security risk if the file is being loaded  
11 from some unknown, untrusted, remote server. As will be described in  
12 conjunction with blocks 820-834, this security risk is overcome by utilizing the  
13 present method for transferring objects between remote boundaries.

14 At block 812, the cmdlet is executed. Execution includes instantiating the  
15 cmdlet class to create a cmdlet object. Populating the properties specified in the  
16 cmdlet object in the manner specified in the cmdlet class. Creating objects as  
17 defined in the code of the cmdlet class. If the incoming objects (such as objects  
18 from a previous cmdlet in a pipelined command) do not match the type specified  
19 in the current cmdlet class, the extended type manager may coerce the type as  
20 needed. Processing continues at block 814.

21 At block 814, the process may manipulate the created objects by accessing  
22 any of the properties and by invoking any of the methods. These created objects  
23 are referred to as "live" or "raw" objects because the object's methods are  
24 available and can be invoked.

1       As one can imagine, the processing described in blocks 808-814 is less than  
2 ideal when the cmdlet is located on a remote computer. If the remote file was  
3 loaded into the local process for execution, there is a chance that the remote file  
4 could execute malicious code, which may compromise the security of the  
5 requesting entity. In addition, in an administrative tool environment, the  
6 requesting entity (e.g., administrator) is generally interested in obtaining  
7 administrative information regarding the remote computer. Thus, the objects  
8 populated by the cmdlet must contain status or other information pertinent about  
9 the remote server and not the requesting entity (e.g., administrator).

10       As explained in the background section, in a traditional web service  
11 environment, the client is responsible for knowing and trusting the remote server  
12 to whom it wishes to communicate. However, as explained above, this greatly  
13 restricts to whom the client may communicate because only objects that support a  
14 specific interface may communicate with the requesting entity. The present  
15 method for communicating objects between a requesting entity (e.g.,  
16 administrator) and a remote server does not impose these limitations and is now  
17 described. For convenience, blocks which perform processing on the remote  
18 entity (e.g., remote server or separate process) are illustrated with a "dotted"  
19 background.

20       Returning to decision block 806, if the cmdlet is to be run in another  
21 process or on a remote server, processing continues at block 818. At block 818,  
22 the client establishes communication with a remote agent associated with the  
23 remote entity. The remote agent then performs blocks 820-824, which performs  
24 the functioning described above for blocks 810-812, but instead performs them  
25



1 with respect to the remote entity. However, once the cmdlet is executed and has  
2 obtained its associated objects, processing continues at block 826.

3 At block 826, the objects are serialized in a manner such that the security of  
4 the requesting entity (e.g., administrator) is not compromised. The serialization  
5 process creates a return package, such as return package 730 illustrated in  
6 FIGURE 7. In overview, serializing the objects serializes information in a manner  
7 that can not be harmful to the requesting entity. An overview of the serialization  
8 process is illustrated in FIGURE 9 and a more detailed description of the  
9 serialization process is described below in conjunction with FIGURE 10.  
10 Processing continues at block 828.

11 At block 828, the serialized objects are transmitted to the client computer.  
12 The transmission of the serialized objects may be performed using convention  
13 methods known for network communication or may use known inter-process  
14 communication if the requesting entity and the remote server are on the same  
15 computer. Processing continues at block 830.

16 At block 830, the serialized objects are received at the requesting entity.  
17 The serialized object type is a type registered in the extended type manager. Thus,  
18 upon receipt, the extended type manager recognizes the serialized object type.  
19 Processing continues at block 832.

20 At block 832, the serialized object is de-serialized into sub-component  
21 objects. Briefly, the de-serialization process decomposes the serialized object into  
22 known base types. For any property that is not a base type, the de-serialization  
23 keeps the property as a property bag. An overview of the de-serialization process  
24 is illustrated in FIGURE 9 and a more detailed description of the de-serialization  
25

1 process is described below in conjunction with FIGURE 11. Processing continues  
2 at block 834.

3 At block 834, the de-serialized objects that were one of the base types may  
4 be manipulated as “raw” objects, meaning that the methods and properties  
5 associated with the object are available. De-serialized objects that are not one of  
6 the base types may not be manipulated as “raw” objects. These objects are  
7 referred to as “deserialized” objects, which mean that the objects contain  
8 information about certain properties for the object, but their methods are not  
9 available.

10 FIGURE 9 is a block diagram illustrating a general overview of the  
11 serialization process and de-serialization process suitable for use in FIGURE 8. In  
12 general, the serialization process 900 converts an object 902 that was created by a  
13 cmdlet executing on the remote entity into a hierarchical tree 904. In general, the  
14 hierarchical tree 904 is an object that represents object 902 broken down into sub-  
15 components. Some of the sub-components are object types known by both entities  
16 (e.g., the requesting entity and the remote entity). These known types are  
17 identified in the list of base types 712 and 722 on the requesting entity and remote  
18 entity, respectively (see FIGURE 7). When base types 712 and 722 do not contain  
19 the same list of base types, a protocol negotiation process, described in detail  
20 below in conjunction with FIGURE 13, is performed to determine which list of  
21 base types is used. Thus, as will be explained in greater detail below in  
22 conjunction with FIGURE 10, each entry within the hierarchical tree is serialized  
23 and added to the serialized object 906. In one embodiment, the serialized object  
24 906 comprises an XML document. This serialized object, along with other  
25

1 information, is combined to form the request package that is transmitted to the  
2 requesting entity.

3       Upon receiving the request package, the requesting entity retrieves the  
4 serialized object 906 and performs a de-serialization process 910. In general, the  
5 de-serialization process 910 reconstructs the hierarchical tree 904. Because the  
6 hierarchical tree 904 is one of the object types recognized by the extended type  
7 manager, the extended type manager de-composes the hierarchical tree 904 into  
8 base objects and/or property bag objects 912. The base objects are “live” (i.e.,  
9 “raw”) objects that have functionality (e.g., methods). The property bag objects  
10 912 (i.e., “de-serialized” objects) provide relevant data, but do not include  
11 methods. Thus, as this overview illustrates, the object 902 originally created on  
12 the remote entity has been transferred to the requesting entity as a plurality of  
13 “raw” base objects and other “de-serialized” objects without compromising the  
14 security of the requesting entity or requiring any special interface for the object  
15 902. This de-serialization process 910 is described in more detail in conjunction  
16 with FIGURE 11.

17       FIGURE 10 illustrates an exemplary process 900 for serializing objects. As  
18 described above, as a general overview, the serialization process 900 decomposes  
19 an object and its properties (properties are also objects) into known base types and  
20 possibly into one or more property bags. The process 900 is recursive in nature  
21 and continues until the objects (including properties) have either been decomposed  
22 into one of the base types, or until each of the remaining objects is not listed in the  
23 known base types. Because the serialization process 900 is recursive and may  
24 create a hierarchical tree that is difficult to manage, process 900 may also be  
25 limited using several different techniques that will be described below. In order to

1 better explain process 900, the following discussion uses an example object (e.g.,  
2 SQLInfo object) hypothetically created by the "get/DB" cmdlet. In general, the  
3 SQLInfo object may be defined in any conventional manner. The following is one  
4 of those conventional manners:

```
5
6     SQLInfo
7     {
8         public int EmployeeNumber
9         [Attribute("Identification")]
10        public string Name;
11        public residence Address;
12        public int SSN;
13        public DateTime BirthDate;
14    }
```

15  
16 FIGURE 12 illustrates a graphical representation of a portion of a  
17 hierarchical tree that is created during process 900. The hierarchical tree  
18 represents a decomposed portion of the SQLInfo object shown above and will be  
19 used in conjunction with FIGURE 9 to describe the serialization process.

20 Process 900 begins at block 1002, where an object created by a remote  
21 cmdlet has been created. At block 1002, a protocol negotiation process is  
22 performed in order to identify the level at which the requesting entity and the  
23 remote entity may communicate. Briefly, described in detail later in conjunction  
24 with FIGURE 13, the level at which the entities communicate determines the  
25 amount of information that is transmitted to the requesting entity in order to

1 transfer the object. As will be explained, the protocol negotiation process allows  
2 both the requesting entity and the remote entity to upgrade their system  
3 independent of each other while still allowing them to communicate with each  
4 other using different versions. In practice, this allows a requesting entity, which is  
5 operating with a version of base types that is several years newer than the version  
6 on the remote entity, the ability to still communicate with the remote entity. The  
7 converse is also true. Therefore, systems that have been shipped many years ago  
8 can be managed by newer administrator computers. The output of the negotiation  
9 process is a negotiated list that identifies each of the negotiated base types. Once  
10 the level of communication is determined, processing continues at block 1004.

11 At block 1004, a property bag is created for holding information about the  
12 object. In one embodiment, the property bag may be implemented as a hash table.  
13 The property bag is a core data type supported by the administrative tool  
14 environment. Processing continues at block 1006.

15 At block 1006, the type of object is identified by reflecting on the object.  
16 Once the type is identified, processing continues at decision block 1008. At  
17 decision block 1008, a determination is made whether the negotiated base types  
18 include the identified type of object. This determination may be implemented as a  
19 look-up within the negotiated list. For this implementation, the negotiated list  
20 would identify each of the negotiated base types. If the object is identified as one  
21 of the negotiated base types, processing continues at block 1010.

22 At block 1010, an entry is created in the property bag for the identified  
23 object. An exemplary implementation of a property bag will now be described.

24 Turning briefly to FIGURE 12, a graphical depiction of a portion of an  
25 exemplary hierarchical tree (e.g., property bag tree) is illustrated. The portion

1 depicts two levels within the hierarchical tree: the first level having property bag  
2 1220; a second level having sub-property bag 1240. Each property bag 1220 and  
3 1240 includes a name field 1202, a value field 1204, and a type field 1206. In  
4 addition, each property bag 1220 and 1240 may include an IsType field 1208, a  
5 WasType field 1210, and a TreatAs field 1212. Property bag 1220 includes entries  
6 1222-1230 and sub-property bag 1240 includes entries 1242-1248.

7       Returning to FIGURE 10, as mentioned above for block 1010, an entry for  
8 the object is created in property bag 1220 (e.g., entry 1222). Each field for the  
9 entry is then updated with information relating to the object. For example, the  
10 name for the object (e.g., "EmployeeName") is entered in the name field 1202. A  
11 value (e.g., 28731) for the property is entered in the value field 1204. The type of  
12 the object (e.g., "Int") is entered in the type field 1206. In one embodiment, the  
13 name field may be a key for the hash table. Processing continues at block 1012.

14       At block 1012, the object is serialized. Direct serialization may occur via  
15 mechanisms provided by the underlying component framework, or by code that is  
16 specified for the base type, or the like. The serialization process for the object  
17 depends on its base type. Each base type has its own special serialization process.  
18 In one embodiment, the special serialization process is identified within the  
19 negotiated list. For this embodiment, the negotiated list includes a reference to a  
20 special serialization process for each of the negotiated base types in the list. In  
21 one embodiment, direct serialization converts the object into XML. However,  
22 direct serialization may convert the object into formats other than XML without  
23 departing from the appended claims. The result of the special serialization process  
24 is stored within the serialized object.

1       Returning to decision block 1008, if it is determined that the object is not  
2 identified as one of the negotiated base types, processing continues at block 1014.  
3 At block 1014, a property bag type is entered in the type field 1206. This property  
4 bag type then references a sub-property bag (e.g., sub-property bag 1240) created  
5 in block 1016.

6       At block 1016, a sub-property bag (e.g., sub-property bag 1240) is created.  
7 Thus, by creating additional levels of sub-property bags, a property bag tree is  
8 formed. The sub-property bag has the same fields as defined for the property bag.  
9 This sub-property bag may also have an entry for one or more sub-property bags  
10 creating another level in the hierarchical tree of property bags. Processing  
11 continues at block 1020.

12       At block 1020, the recursive nature of the serialization process is  
13 graphically depicted. The serialization process focuses on serializing each desired  
14 property within the object. Typically, the desired properties are properties that are  
15 designated as public. However, in some cases, hidden properties and other  
16 properties may be desired properties, and, are also serialized. Blocks 1022-1032  
17 may be performed in various orders and in some cases one or more of the blocks  
18 (e.g., block 1032) may be optionally performed. Each of the blocks 1022-1032 is  
19 now further described.

20       At block 1022, a type for the desired property is identified. This again  
21 occurs via reflection as explained above in reference to block 1006.

22       At block 1024, an entry into the current property bag (e.g., a sub-property  
23 bag) is added for the property.

24       At block 1026, the fields in the entry associated with the desired property  
25 are set. The fields are set as explained above for block 1010, but are set in the

1 sub-property bag, rather than the property-bag. For example, the fields for entry  
2 1242 in sub-property bag 1240, shown in FIGURE 12, are updated with  
3 information associated with the property (object).

4 At block 1028, if the desired property is one of the negotiated base types,  
5 the property is directly serialized as described above for block 1012.

6 At block 1030, if the desired property is not one of the negotiated base  
7 types, blocks 1014 and 1020 are performed in a recursive fashion.

8 Due to this recursive process, the hierarchical tree of property bags may  
9 become difficult to manage. Thus, it may be desirable to limit the serialization  
10 process 900. Embodiments for limiting the serialization process are represented  
11 within block 1032, and may optionally be performed. There are several  
12 embodiments for culling the hierarchical tree. In one embodiment, a policy may  
13 be set that specifies a predetermined depth of the hierarchical tree. For example, if  
14 the predetermined depth is set at two, the recursive nature of the serialization  
15 process stops after creating one or more sub-property bags referenced from the  
16 property bag. The serialization process 900 does not proceed any further with  
17 decomposing the objects within the sub-property bags. Thus, these sub-property  
18 bags are serialized as a property bag and are included within the serialized object.

19 In another embodiment, prior to serialization, the remote agent runs the  
20 object through a "pick" process that strips out the properties that were not  
21 specified. An exemplary syntax is as follows: >Rcmd machine:RemoteServer  
22 Get/DB | Pick Name, Birthdate. Then, during serialization, the specified  
23 properties (e.g., Name and Birthdate) become the desired properties and are  
24 serialized as described above. The pick process may be another cmdlet that is  
25 available on every system. One disadvantage of explicitly specifying the



1 properties with the pick process is that the pick list must be specified each time  
2 and that the desired properties must be known in order to pick them.

3 In still another embodiment, block 832 may use a property set mechanism.  
4 The property set mechanism allows a name to be defined for a set of properties.  
5 An administrator may then specify the property set name, along with the cmdlet,  
6 in order to obtain the desired set of properties. The property set may be defined in  
7 various ways. In one way, a predetermined parameter, such as "?", may be entered  
8 as an input parameter for a cmdlet. The operating environment upon recognizing  
9 the predefined parameter may list all the properties of the object. The list may be  
10 a GUI that allows an administrator to check (e.g., "click on") the desired  
11 properties and name the property set. The property set information is then stored  
12 in the extended metadata. For example, a property set may be named  
13 "performance". Each object would then identify which of its properties should be  
14 included in this performance property set. Therefore, the administrator does not  
15 need to know the name of the properties, but rather only needs to know the name  
16 of the desired property set. A property set named "default" may be defined that  
17 specifies the desired properties to serialize for an object. The administrator could  
18 then specify the default property when serialization is necessary.

19 Another way of limiting the property bag hierarchical tree is by only  
20 supporting certain types and forcing types derived from the certain types to  
21 conform to that type. In one implementation, the object hierarchy is "walked  
22 down" by reflecting on each object to identify its parent. For certain objects, if it  
23 derived from a base type, the object will be forced to be its parent type. For  
24 example, a hash table object is an Idictionary object, which is an IEnumerable  
25 object, which derives from a base object. The serialization process may treat the

1 hash table object as an IEnumerable object. In this way, the serialization process  
2 completely supports objects at certain levels within the object hierarchy, but forces  
3 objects of other types within the object hierarchy to conform to the chosen type.  
4 In practice, even though the original object type is not transferred, once the  
5 information is obtained at the receiving computer, the receiving computer can  
6 perform processing on it (such as creating a key) to obtain the original object, if  
7 needed.

8         Once the serialization process 900 is completed, the serialized object, along  
9 with other information, is combined to form the request package that is transmitted  
10 to the requesting entity.

11         The property bag and sub-property bags may have additional fields that are  
12 set during the serialization process, such as the IsType field 1208, the WasType  
13 field 1210, or the TreatAs field 1212. These types are used during the de-  
14 serialization process 910. The IsType field 1208 identifies a type for the object.  
15 However, when the object is serialized, the IsType field 1208 for the object  
16 becomes "NULL". The WasType field 1210 identifies the type the property was  
17 before changing. From this information, the de-serialization process 910 can  
18 identify where the object originated, but can not know how many properties the  
19 object currently has. For example, if a property set or pick process was used when  
20 serializing the object, only specific properties were serialized, not all. The TreatAs  
21 field 1210 identifies a type and guarantees that the properties associated with that  
22 type are available. For example, using the property set example above, one could  
23 define a new object type as "SQLInfoPersonal". This new type guarantees that  
24 specified properties associated with the object are available, such as home address,  
25 home phone number, and the like. If sufficient properties are specified to include

1 all the properties needed to qualify as a TreatAs, the serialization process 900 will  
2 enter the correct type under the TreatAs field 1210.

3 FIGURE 11 is a logical flow diagram illustrating an exemplary process for  
4 de-serializing objects suitable for use in the processing of the cmdlet shown in  
5 FIGURE 8. In general, de-serializing the transmitted objects performs the  
6 converse operations as the serialization process. Processing begin at block 1102,  
7 where a property bag indicator is identified within the transmitted data. In one  
8 embodiment, the indicator may be an element within an XML document.  
9 Processing continues at block 1104.

10 At block 1104, a property bag object is created that will be populated with  
11 information contained within the transmitted data.

12 At block 1106, a type property for the property bag object is set. The type  
13 property informs the extended type manager so that the extended type manager  
14 can handle the property bag as needed.

15 At block 1108, each element identified within the transmitted data and  
16 associated with the property bag is processed. At block 1110, an entry within the  
17 property bag is created. At block 1112, the fields within the property bag are  
18 populated. Processing continues at decision block 1114.

19 At decision block 1114, a determination is made whether another property  
20 bag is defined within the transmitted data. If there is another property bag defined  
21 within the transmitted data, processing loops back through blocks 1102-1108 until  
22 the transmitted data does not contain any more property bags.

23 However, once there is no more property bags, processing continue at  
24 blocks 1120-1140. Once the hierarchical tree of property bags has been created,  
25 each property bag is reviewed for further processing.

1       At block 1120, if the type property for the property bag is one of the base  
2 types, the de-serialization process instantiates an object of the identified type  
3 (block 1122) and populates the properties of the object (block 1124).

4       At block 1130, if the TreatAs attribute for the property bag specifies a type,  
5 an object of the specified type is instantiated (block 1132) and properties of the  
6 object are populated (1134). The type specified in the TreatAs attribute is one of  
7 the base types.

8       At block 1140, for each remaining property bag that is not of a base type,  
9 each object identified within the property bag is instantiated (block 1142) and  
10 populated (block 1144).

11       Thus, as one can see, if the original object created at the remote computer is  
12 a base type, the transmitted data includes minimal information. This corresponds  
13 with a hierarchical tree with only one level. The transmitted data would include  
14 the name/value pairs for the properties but would not contain any executable code.  
15 However, once the requesting computer received the transmitted data and  
16 recognized the type, an object of that type would be instantiated and would have  
17 the methods and functionality provided by the original object. In this manner,  
18 malicious code is not transmitted to and executed by the requesting computer.  
19 Instead, non-harmful information about the object is transmitted to the requesting  
20 computer.

21       FIGURE 13 is a logical flow diagram illustrating an exemplary process for  
22 negotiating a protocol suitable for use in conjunction with the process for  
23 serializing objects shown in FIGURE 9. Even with limiting the specific  
24 parameters to serialize or specifying a depth for the hierarchical tree, for certain  
25 objects, the amount of information that is transferred between the two computers

1 is quite large. Thus, an additional negotiation mechanism is provided that limits  
2 the amount of information that is necessary for transmission. As described above,  
3 the complexity of the hierarchical tree depends on the number of negotiated base  
4 types. When a large number of negotiated base types are known, the hierarchical  
5 tree becomes less complex, which ultimately results in less information  
6 transmitted to the requesting computer. For any of the entries within the property  
7 bag specifying a base type, only minimal information is transferred in order to  
8 create a "raw" object on the requesting computer. Thus, it is envisioned that the  
9 number of base types specified within the list of base types may grow in order to  
10 accommodate more and more objects. However, the present method still does not  
11 place artificial restrictions or requirements on software developers in order to use  
12 this method. For example, in some prior environments, both computers had to be  
13 updated with the same version in order to operate properly. In this prior  
14 environment, one computer dictated which computers could communicate with it  
15 and forced compliance on the other computers in order for communication to  
16 occur. In contrast, the present method of transferring objects provides a protocol  
17 negotiation method that allows computers to update independently of each other  
18 and still be able to communicate with each other.

19 The protocol negotiation process 1300 begins at block 1302. At block  
20 1302, a client version number is received. The client version number identifies the  
21 most recent version of the list of base types that the client (e.g., requesting  
22 computer) may use. Processing continues at decision block 1304.

23 At decision block 1304, the client version number is compared with the  
24 remote version number to determine whether the remote version number is more  
25 recent. The remote version number identifies the most recent version of the list of

1 base types that the remote server may use. If the remote version number is more  
2 recent, processing continues at block 1306.

3 At block 1306, the list of base types associated with the client version  
4 number is used for the serialization process. Thus, the client version becomes the  
5 negotiated list that is used during the serialization process.

6 On the other hand, if the remote version number is not as recent as the  
7 client version, processing continues at block 1308. At block 1308, the most recent  
8 list of base types available on the remote computer is used for the serialization  
9 process. Thus, the remote version becomes the negotiated list that is used during  
10 the serialization process.

11 In another embodiment of the negotiation process, the requesting computer  
12 may send the base types that it supports to the remote computer. The remote  
13 computer may then walk through the table and decide which types are supported  
14 by accepting or rejecting items in the table. The types that are accepted then form  
15 the negotiated list.

16 In another embodiment of the negotiation process, the requesting computer  
17 may send a set of references. The set of references may specify a file name. Thus,  
18 any type of object within the specified file name would be supported and be one of  
19 the negotiated base types.

20 Although details of specific implementations and embodiments are  
21 described above, such details are intended to satisfy statutory disclosure  
22 obligations rather than to limit the scope of the following claims. Thus, the system  
23 and method described above is defined by the claims is not limited to the specific  
24 features described above. Rather, the present system and method is claimed in any  
25

1 of its forms or modifications that fall within the proper scope of the appended  
2 claims, appropriately interpreted in accordance with the doctrine of equivalents.  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14  
15  
16  
17  
18  
19  
20  
21  
22  
23  
24  
25